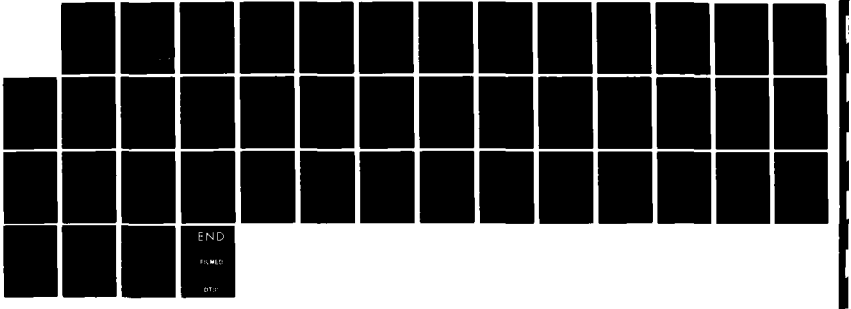
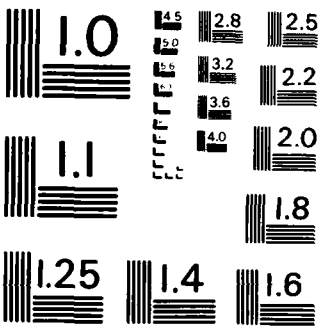
AD-A161 429 FUNCTIONAL TESTING OF LSI/VLSI (VERY LARGE SCALE INTEGRATION) DIGITAL SYSTEMS(U) THOMAS J WATSON SCHOOL OF ENGINEERING APPLIED SCIENCE AND TEC.. S Y SU ET AL. 1/1

UNCLASSIFIED 30 AUG 84 DAAB07-82-K-J056 F/G 9/5 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS - 1963 - A

FINAL TECHNICAL REPORT

Functional Testing of LSI/VLSI Digital Systems

August 30, 1984

Contract No. DAAB 07-82-K-J05-6

Submitted to

Headquarters, United States Army
Communication Electronic Commands
Fort Monmouth, New Jersey 07703

Prepared by

Stephen Y.H. Su
Tony Sheng Lin
Li Shen

Research Group on Design Automation
and Fault-Tolerant Computing
Thomas J. Watson School of Engineering,
Applied Science and Technology
Binghamton, New York 13901

AD-A161 429

DTIC FILE COPY

DTIC
ELECTE
NOV 2 0 1985
S D
A

This document has been approved for public release and sale; its distribution is unlimited.

85 8 8 054

SUMMARY

Due to the advances in the integrated circuit (IC) technology, more and more components are being fabricated into a tiny IC chip. Since the number of pins on each chip is limited by the physical size of the chip, the problem of testing becomes more difficult than ever, especially in the VLSI (Very Large Scale Integration) chips. This problem is aggravated by the fact that, in nearly all cases, integrated circuit manufacturers are not willing to release the detailed circuit diagram of the IC chip to the users. Yet, as users of the IC chips, to make sure that the implemented system is reliable, we need to test the IC chips and the systems made of the interconnection of these chips. The purpose of this project is to find efficient algorithms for testing LSI/VLSI chips and LSI/VLSI-based systems.

As a result of the rapidly increasing complexity of modern digital LSI/VLSI systems, functional testing is attracting more attention than ever not only in the computer manufacturing industry but also in the diversified potential applications.

Functional testing uses a representation of a digital system higher than the gate-level testing. In functional testing, functional faults with respect to the functional specification (e.g., addition operation in a processor) are tested instead of a signal faults (e.g., a line stuck-at logical 0) in the circuit representation. The purpose of functional testing is to validate correct functional operations of digital systems according to their specifications. Using functional testing techniques, one cannot only reduce the test generation complexity but also obtain a test set for testing the digital systems with the same functions but different circuit design/implementation (e.g., parallel adder vs. serial adder).

Using RTL, the behavior of a microprocessor is comprehensively described, and functional faults derived from them can be studied. In [1], two approaches

for functional testing are given based on the RTL description. The first approach constructs a data graph from the RTL description and uses the existing algorithm such as the D-algorithm or path sensitizing method to generate the tests for functional faults. In the second approach, the symbolic simulations technique is used to generate tests for detecting faults in the control signals. In [2], a formal definition of RTL is defined as:

$$k: (t,c)\ R_d \leftarrow f(R_{s1}, E_{s2}, \ldots, R_{sv}), \rightarrow n$$

where,

k is the statement label

t is the timing and c is the condition to execute the statement

R_d is the destination register

R_{si} is the ith source register

f is an operation on R_{si}

$\leftarrow$ represents data transfer

$\rightarrow$ n represents a jump to statement n

Accession For
NTIS GRA&I
DTIC TAB
Unannounced
Justification
By
Distribution/
Availability Codes
Dist | Avail and/or Special
A-1

For example, the following RTL statement No. 17: $(T_5\ C_8)\ R_2 \leftarrow R_3 + R_5, \rightarrow 38$ means that when $T_5 = C_8 = 1$, the sum of R_3 and R_5 will be stored in R_7 and then the program jumps to statement No. 38.

Based on the above notation, eight catagories of fault can then be identified as timing faults (t/t'), condition faults (c/c'), register decoding faults (R_i/R_i'), instruction decoding (function selection) faults (f/f'), control faults (n/n'), data storage faults ($(R_i)/(R_i')$), data transfer faults ($\leftarrow/\leftarrow'$) and data manipulation (function execution) faults ((f)/(f')). This set is functional comprehensive because the behavior of a CPU can be described by a sequence of RTL statement. Three procedures for testing those five fault catagories (except timing, condition, and control faults) are derived. The testing requires the creation of executable sequences to form a "sensitizing" path which leads from a faulty statement to a statement producing faulty output

QUALITY INSPECTED 3

information. The RTL technique seems to be a promising approach for functional testing.

Recently, we presented three algorithms to test the instruction decoding function of microprocessors [3]. The algorithms are based on the knowledge of some timing and control information available to users through microprocessor manuals and data sheets. The tests are functional in nature. We establish the order of complexity of the algorithms presented in this paper. As an example, the test complexity for a microprocessor is computed and the results are compared with a known algorithm.

In [4] we present the state-of-the-art for the functional testing of LSI/VLSI devices with special emphasis on microprocessor testing. Various types of IC chips are briefly discussed. Different approaches for testing the functional faults of LSI/VLSI are surveyed and the comparison of these methods are given. Fault models for representing the faults and fault coverage of the tests are discussed. Some of the important unsolved problems and current trends in testing VLSI are pointed out.

A new approach for testing VLSI circuits is presented in [5]. Through backward critical path tracing, a test and all faults detectable by the test are generated simultaneously. Therefore, the expensive fault simulation is completely eliminated. We present a critical path test generation procedure for digital systems described by hardware description language (HDL). A multiplication circuit described by a HDL is utilized for demonstrating the test generation method.

In this report, two functional testing techniques are presented with in-depth technical discussion. The first technique (Part I) provides a functional testing method for microprocessors. Major issues in testing microprocessors are clarified and defined. The second technique (Part II) is a new algorithm to systematically perform functional test generation for digital LSI/VLSI systems using machine symbolic execution technique. Skills and concepts developed in the area of artificial intelligence (AI) are applied.

The first technique (Part I) is for testing microprocessors [6]. Among the variety of LSI/VLSI devices, microprocessors have the most widespread use and the highest functional complexity. Therefore, recently, testing microprocessors has received a great deal of attention. Several deterministic testing methods have been proposed. The more important approach to the functional testing of microprocessors is the Thatte and Abraham's method which has been widely cited in subsequent literature, but their fault model for the instruction execution needs to be generalized. For example, in an instruction decoding fault I_j/I_j+I_k, it is assumed that instead of execution I_j, both instructions I_j and I_k are executed to completion. This is not general. In order to make the fault model more general and practical, partial execution of an instruction under fault should be considered. In addition, a microprocessor is a type of complex sequential machine. The current approach is to test microprocessors by instruction execution. Generally, before executing an instruction-under-test, we have to write certain data into some registers, and after extending the instruction, read the contents of the registers. Therefore, if the write or the read instruction is faulty, we may not be able to test the instruction-under-test. To solve this problem, Thatte and Abraham have to label instructions and define test order in detail before testing. This makes the test procedure more complex.

In our work, we first establish a fault model for microprocessors, emphasizing the control fault model defined at the register transfer language (RTL) level, since it is convenient to represent the instruction decoding faults and other control faults at such a level. Then we consider the basic instructions for the write and read register functions as the kernel of microprocessor. This kernel can be represented by a sequential machine. Based on the fault model, we

can use checking experiment to verify the kernel. Thus, testing microprocessor is divided into tow steps, i.e. guarantee the correctness of the kernel first, then use the kernel for testing each instruction. Therefore, the complexity of test generation will be reduced.

The second technique (Part II) is based on two major foundations [7]. First, after the standard syntax of a register transfer language is defined. a register transfer level fault model is developed. All types of faults covered by the fault model were analyzed and the number of faults was reduced. Secondly, based on the RT-level fault model derived, the technique of symbolic execution was employed. Symbolic execution is a kind of program execution technique which manipulates symbolic variables instead of variable values during program execution. In A.I., this technique is intensively used for automatic theorem proving. program verification, programming in logic and many other interesting topics. Since test generation of LSI/VLSI systems is also one of the important issues in A.I. applications, the symbolic execution technique which is popular in A.I. application was adopted. This powerful technique seems to provide a promising solution for future testing problems of digital LSI/VLSI systems.

REFERENCES

[1] S. Su & Y. Hsieh, "Testing functional faults in digital systems described by register transfer language", J. of Digital Systems, Also, Digest of papers, 1981 Test Conf., pp. 447-457.

[2] Y. Min and S. Su, "Testing functional faults in VLSI", Proc. of 19th Design Auto. Conf., 1982.

[3] K. Saluja, L. Shen & S. Su, "A simplified algorithm for testing microprocessors", 1983 Test Conf., pp. 608-675.

[4] S.Y.H. Su and T. Lin, "Functional testing techniques for digital LSI/VLSI systems," Invited paper, Proc. of 21st Design Automation Conference, Albuquerque, NM, June 25-27, 1984, pp. 517-528.

[5] L. Shen and S.Y.H. Su, "VLSI functional testing using critical traces at a hardware description language level," Proc. of the second GI/NTG/GMR Conf. on fault-tolerant Computing Systems, GMD, Bonn, West Germany, Sept. 19-21, 1984.

[6] L. Shen and S. Su, "A functional testing method for microprocessors", Proc. 14th International Symp. on Fault-tolerant Computing, June, 1984.

[7] T. Lin and S.Y.H. Su, Functional test generation of digital LSI/VLSI systems using machine symbolic execution techniques," Digest of Papers, 1984 International Test Conference, Philadelphia, PA, Oct. 16-19.

[8] S. Su, "Computer-aided design and testing of digital systems and circuits", (invited paper), Proc. Jordan Int'l Elect. & Electronic Eng. Conf., 1983.

PART I

I. INTRODUCTION

The development of integrated circuit technology has resulted in a wide range of applications for microprocessors. Testing of microprocessors is a difficult problem because of the complexities of microprocessors. The problem is more serious for users due to lack of information on internal implementation of microprocessors and other VLSI chips. Recently, several deterministic testing methods have been proposed to solve this problem. These testing techniques are essentially based on functional level [1-11].

A microprocessor is a type of complex sequential machine. The current approach is to test microprocessors by instruction execution. Generally, before executing an instruction-under-test we have to write certain data into some registers, and after executing the instruction, read the contents of the registers. Therefore, if the write or the read instruction is faulty, we may not be able to test the instruction-under-test. To solve this problem, Thatte and Abraham [3] have to label instructions and define test order in detail before testing. However, they do not consider the partial execution of an instruction. So for instruction decoding fault $I_j/I_j + I_k$, it is assumed that instead of executing I_j, both instructions I_j and I_k are executed to completion. It is more general and practical to consider partial execution of an instruction under fault. Our fault model allows this.

Abraham and Parker [5] proposed a simplified fault model. First, one tests all internal registers, then executes all instruction and data manipulation functions.

In this paper, we consider the basic instructions for the write and read register functions as the kernel of a microprocessor. This kernel can be represented by a sequential machine. Based on the fault model, we use checking experiment

to verify the kernel. Then we use the kernel for testing each instruction. The control fault model is established at the Register Transfer Language (RTL) level, since it is convenient to represent the instruction decoding faults and other control faults at such a level.

Section II presents a fault model for microprocessors, emphasizing the control fault model defined at the RTL level instead of the instruction level. In Section III, after examining most existing off-the-shelf microprocessors, we derive testing requirements based on different types of operations. In Section IV, we define the write and read sequences as the kernel of a microprocessor. Then Section V proposes a kind of test data which is quite powerful. Section VI presents the verification of the write and read sequences. Section VII discusses the testing of control faults. Finally, conclusions are given in Section VIII.

II. FAULT MODEL

The functions of a microprocessor are mainly performed by instruction execution. The sequence of operations for an instruction can be described by RTL. We consider that an instruction consists of a series of RTL statements. The typical statement is defined as

(conditions): $D \leftarrow f(S_1, S_2, \ldots, S_i, \ldots)$

where

D - destination

S_i - Source

$f(S_1, S_2, \ldots, S_i, \ldots)$ - operation

Destinations and sources may be internal registers of a microprocessor or external to the microprocessor (i.e., data bus, address bus, etc.). We are only concerned with those internal registers which are of interest to users, so we do not consider implied registers such as buffers. For example, data transfer from memory to memory can be described as $DB_j \leftarrow DB_i$, instead of $Buffer \leftarrow DB_i$ followed by $DB_j \leftarrow Buffer$, where DB denotes the data bus which represents data input or output of memory, i,j denote different bus cycles, DB_i (read from memory) is ahead of DB_j (write into memory).

After examining most existing off-the-shelf microprocessors, e.g. Intel 8080 and 8086, Zilog 80 and 8000, Motorola 6800 and 68000, the RTL-like operations can be divided into two classes, transfer operations (class T, $D \leftarrow S$), and arithmetic and logical operations (class A). Class A can be subdivided into six subclasses based on the combination of destinations and sources as shown in Table 1, where the content of flag bits constitute a status register.

Class	Type of Expression	Operation
A1	$D \leftarrow f(D)$	BIT SET BIT RESET BIT COMPLEMENT INCREMENT DECREMENT DECIMAL ADJUST SHIFT ROTATE COMPLEMENT NEGATE CLEAR
A2	$D \leftarrow f(D,S)$	ADDITION ADDITION WITH CARRY SUBTRACTION SUBTRACTION WITH BORROW AND OR XOR
	$D \leftarrow f(S)$	EXTEND SIGN
A3	$D \leftarrow f(S_1,S_2)$	ADDITION, $D \leftarrow S_1+S_2$
	$D \leftarrow f(D,S_1,S_2)$	ADDITION, $D \leftarrow D+S_1+S_2$
A4	$D \leftarrow f(S_1,S_2,S_3)$	ADDITION, $D \leftarrow S_1+S_2+S_3$
AM	$D_1,D_2 \leftarrow f(D_1,S)$	MULTIPLY DIVIDE
AF	$Flags \leftarrow f(S)$	BIT TEST
	$Flags \leftarrow f(S_1,S_2)$	COMPARE
	$Flags \leftarrow f(S_1,S_2,\ldots)$	Modifying flags for all arithmetic and logical instructions

Table 1. RTL-Like Operations

Note that the control operations in RTL control statements such as conditional branch are not listed because we can use RTL assignment statements with conditions and expand the RTL description for instructions with loops.

A microprocessor usually can be divided into two sections: the data processing part and the control part [2,11]. One can define faults for each part. In this paper, we emphasize the control faults.

A. Data processing faults

(1) Data storage fault (R)/(R)'

This means that the content of register is changed from (R) to (R)' due to faults such as stuck-at, bridging and pattern sensitive faults.

(2) Data transfer fault ←/←'

The fault occurs in the transfer path between the sources and the destination. This type of fault includes stuck-at, bridging and pattern sensitive faults.

(3) Data manipulation fault (f)/(f)'

This is the operation execution fault. Under this fault, the operation f is executed, but the result of operation is wrong.

B. Control faults

This kind of fault involves register decoding faults, instruction decoding faults and other control faults. A register decoding fault means missing or changing the selected register, or selection of an extra register, denoted by R/¢, R/R', and R/R+R' respectively. For instruction decoding faults, we consider that an instruction can be executed partially. It means missing or changing the selected operation, or selection of an extra operation in RTL.

In this case. the instruction decoding fault may be I_j/ϕ, $I_j/\Delta I_j$, $I_j/\Delta I_k$, I_j/I_k, $I_j/\Delta I_j+\Delta I_k$, $I_j/I_j+\Delta I_k$, I_j/I_j+I_k, and so forth where ΔI means part of instruction I. The other control faults include instruction execution sequence faults, condition faults and so on.

From the above observation, we assert that it is appropriate to represent the control faults at the RTL level. Therefore, we will define the above control faults at such a level. Let f denote $D \leftarrow f(S_1,S_2,\ldots)$, which is an operation on the instruction-under-test, and $f \varepsilon \{f\}$, where $\{f\}$ is the set of RTL operations ot a microprocessor. Let f' denote $D' \leftarrow f'(S_1',S_2',\ldots)$, which is an unexpected (faulty) operation, and $f' \varepsilon \{f\}$.

We now define three classes (i.e. nine subclasses F1, F2, ..., F9) of control faults as follows:

(1) f/ϕ- No operation is executed.

F1. $\underline{f/\phi}$

(2) f/f' - Instead of performing operation f, another operation f' is executed. It contains two subclasses of faults.

F2. $\underline{\delta f/f'}$: Here δ means that the destination registers D and D' are different and the fault is f/f'.

F3. $\underline{\sigma f/f'}$: σ denotes that registers D and D' are the same.

(3) f/f+f' - In addition to operation f, another operation f' is also executed. It can be subdivided as follows.

(3a) Registers D and D' are different.

F4. $\underline{\delta f/f+f'}$: The source register list of f and f' does not include D' and D respectively. We are not concerned with the execution order

of f and f'.

F5. $\delta f/f'f$: The source register list of f includes D'. f' is executed before performing operation f; i.e. the execution order is

1. $D' \leftarrow f'\ (S_1', S_2', \ldots)$
2. $D \leftarrow f\ (S_1, S_2, \ldots, D'^*)$

where register without * denotes its content before executing the operation, register with * denotes its content after executing the operation.

F6. $\delta f/ff'$: The source register list of f' includes D and the execution order is

1. $D \leftarrow f\ (S_1, S_2, \ldots)$
2. $D' \leftarrow f'\ (S_1', S_2', \ldots, D^*)$

(3b) Registers D and D' are the same. When the source register list of f and f' does not include D' and D respectively, if the execution order is f'f, the fault does not affect the execution of f. If the execution order is f f', it is the same as the case with the fault $\sigma f/f'$.

F7. $\sigma f/f'f$: The source register list of f includes D, and the execution order is

1. $D \leftarrow f'\ (S_1', S_2', \ldots)$
2. $D \leftarrow f\ (S_1, S_2, \ldots, D^*)$

F8. $\sigma f/ff'$: The source register list of f' includes D and the execution order is

1. $D \leftarrow f\ (S_1, S_2, \ldots)$
2. $D \leftarrow f'\ (S_1', S_2', \ldots, D^*)$

F9. $\sigma f/f \amalg f'$: Both f and f' are executed at the same time.

$D \leftarrow f\ (S_1, S_2, \ldots)$

$D \leftarrow f'\ (S_1', S_2', \ldots)$

where L denotes logical AND or OR function depending on the circuit implementation. In this case, the final content of the destination D is the result of the

composite value (i.e. the result of the AND or OR function) of f and f'.

Note that the above control faults can occur at any place in an instruction execution sequence. This control fault model can cover register decoding faults, instruction decoding faults (including partially instruction execution), instruction execution sequence faults, etc., since any control fault can always be defined as missing, changing, or extra RTL opeations and will cause registers to have wrong contents.

III. REQUIREMENTS FOR TESTING CONTROL FAULTS

Our purpose is to test the execution of microprocessor instructions. Therefore, the objective of test pattern generation is to find the initial data in registers (test data) needed for testing functions of an instruction. This test data must satisfy certain requirements. From the control fault model given in Section II, we can obtain various requirements for testing control faults.

Let us establish the following notation. For a fault-free operation f, we have

V_i = the value of register i.

VS_i = the value of the operand in source register S_i.

VD = the value of the operand in destination register D.

VD* = the value of the operation result stored in D.

For a faulty operation f', we obtain VS_i', VD' and VD'* instead.

Theorem 1. Control faults T/ϕ, T/T' and T/T+T' can be detected if the data values of registers satisfy the following requirements:

QTT1. $V_i \neq V_j$, $i \neq j$

QTT2. $V_i \; L \; V_j \neq V_i$, $i \neq j$

Proof. We shall prove this theorem by considering the nine fault classes defined in Section II.

(i) For fault F1 (T/ϕ) and F2 (δT/T'), in order to verify transfer operation T, one needs VS $\neq$ VD.

(ii) For fault F3 (σT/T'), the results of T and T' should be different, i.e. VD* $\neq$ VD'*. To obtain this result, we must have VS $\neq$ VS'.

(iii) For fault F4 (δT/T+T') and F5 (δT/T'T), we need only to detect the extra operation T'. Therefore, VS' $\neq$ VD'.

(iv) Fault F6 ($\delta T/TT'$) means that transfer operation D←S is performed first, then D'←D*. In order to detect the extra operation T', one needs $VS \neq VD'$.

(v) Fault F7 ($\sigma T/T'T$) yields D←S' followed by D←D*. Therefore, we obtain the requirement $VS' \neq VD$.

(vi) For fault F8 ($\sigma T/TT'$), we have D←S, then D←D*. This fault does not affect operation T. In order to verify T, we need $VS \neq VD$.

The above six requirements belong to QTT1.

(vii) For fault F9 ($\sigma T/TLT'$), the composite value of both results of T and T' should be different from the correct result of T. i.e. $VSLVS' \neq VS$ which belongs to QTT2.

Q.E.D.

Theorem 2. Control faults T/A' and T/T+A' can be detected if the data values of registers satisfy the following requirements:

QTA1. $V_i \neq V_j$, $i \neq j$

QTA2. $f'_A \neq VS$

QTA3. $f'_A \neq VD'$

QTA4. $VSLf'_A \neq VS$

where f'_A is the result of operation of class A, i.e. $f'_A = f'_A (VS'_1, VS'_2, \ldots)$.

Proof. The proof is similar to the proof for Theorem 1. Since A' instead of T' is performed, we can change VS' to f'_A in the requirements (ii), (iii), (v) and (vii) in the proof for Theorem 1 to obtain the corresponding requirements for Theorem 2.

(i) For F2 ($\delta T/A'$), $VS \neq VD$, (QTA1).

(ii) For F3 ($\sigma T/A'$), $VS \neq f'_A$, (QTA2).

(iii) For F4 ($\delta T/T+A'$) and F5 ($\delta T/A'T$), $f'_A \neq VD'$, (QTA3).

(iv) For F6 ($\delta T/TA'$), it means that D←S first, then $D' \leftarrow f'_A (S'_1, S'_2, \ldots, D^*)$.

We need f'_A $(VS'_1, VS'_2, \ldots, VS) \neq VD'$. Since VS can be selected as any initial data value, it can be considered as one of several source operands. Therefore, we can rewrite f'_A $(VS'_1, VS'_2, \ldots, VS) \neq VD'$ as $f'_A \neq VD'$, (QTA3).

(v) For F7 ($\sigma T/A'T$), $f'_A \neq VD'$, (QTA3).

(vi) For F8 ($\sigma T/TA'$), it implies D←S followed by D←f'_A $(S'_1, S'_2, \ldots, D^*)$. This requires $VS \neq f'_A$ $(VS'_1, VS'_2, \ldots, VS)$. Here VS can be considered as a destination operand VD'. So we rewrite the inequality as $VD' \neq f'_A$, (QTA3).

(vii) For F9 ($\sigma T/TLA'$), $VS L f'_A \neq VS$, (QTA4). Q.E.D.

Theorem 3. Control faults A/¢, A/T' and A/A+T' can be detected if the data values of registers satisfy the following inequalities.

QAT1. $f_A \neq VD$

QAT2. $f_A \neq VS'$

QAT3. $V_i \neq V_j$, $i \neq j$

QAT4. $f_A \neq VD'$

QAT5. f_A $(VS') \neq f_A(VD)$

QAT6. $f_A \; L \; VS' \neq f_A$

where $f_A = f_A$ $(VS_1, VS_2, \ldots)$, $f_A(VD) = f_A(VS_1, VS_2, \ldots, VD)$, $f_A(VS') =$ f_A $(VS_1, VS_2, \ldots, VS')$.

Proof. Since arithmatical and logical operations instead of transfer operations are considered here, we can change VS to f_A in the cases (i), (ii), (iv), (vi) and (vii) of Theorem 1.

(i) For F1 and F2, $f_A \neq VD$, (QAT1).

(ii) For F3, $f_A \neq VS'$, (QAT2).

(iii) For F4 and F5, $VS' \neq VD'$, (QAT3).

(iv) For F6, $f_A \neq VD'$, (QAT4).

(v) F7 ($\sigma A/T'A$) means that D←S' followed by D←$f_A(S_1, S_2, \ldots, D^*)$ which yields $f_A(VS_1, VS_2, \ldots, VS')$. When there is no fault, $f_A(VS_1, VS_2, \ldots, VD)$ is obtained.

Thus the condition for detecting this fault is $f_A(VS_1, VS_2,\ldots,VS') \neq f_A(VS_1,VS_2,\ldots,VD)$, (QTA5).

(vi) For F8, $f_A \neq VD$, (QTA1).

(vii) For F9, $f_A \; L \; VS' \neq f_A$, (QTA6). Q.E.D.

Theorem 4. Control faults A/A' and A/A+A' can be detected if the data values of registers satisfy the following inequalities.

QAA1. $f_A \neq VD$

QAA2. $f_A \neq f'_A$

QAA3. $f'_A \neq VD'$

QAA4. $f'_A(f_A) \neq VD'$

QAA5. $f_A\,(f'_A) \neq f_A(VD)$

QAA6. $f'_A\,(f_A) \neq f_A$

QAA7. $f_A \; L \; f'_A \neq f_A$

where $f'_A(f_A) = f'_A(VS'_1, VS'_2,\ldots,f_A)$, $f_A(f'_A) = f_A(VS_1,VS_2,\ldots,f'_A)$.

Proof. For the same reason, we may change VS to f_A, and VS' to f'_A for cases (i) to (iii) and (vii) in Theorem 1 to obtain QAA1 to QAA3 and QAA7 respectively. In addition, since the results of A and A' may affect each other, we can obtain QAA4 to QAA6. Q.E.D.

Note that for requirement QAA7, if operation f_A and f'_A of class A are executed in the same unit (e.g. ALU), then both results of f_A and f'_A can no longer be considered as obtained separately. Instead, QAA7 may be considered as a data manipulation fault $(f_A)/(f_A)'$.

IV. WRITE AND READ SEQUENCES

As a microprocessor is one type of sequential machine and all internal registers are memory elements of the sequential machine, the content of registers represents the state of the sequential machine. Therefore, the following general procedure is utilized for testing microprocessors.

1. Initialization of state of registers.
2. Execution of the instruction-under-test.
3. Read the state of registers.

In fact, Steps 1 and 3 consist of write and read register sequences respectively. Obviously, if we can guarantee the correctness of Steps 1 and 3 first, then the testing problem will be simplified.

The testing approach used here is a kind of open loop testing [5]. It implies the use of a test equipment which provides the stimuli to the microprocessor and observes the responses from the microprocessor.

Now let us discuss write and read sequences which are used for writing and reading the register states of a microprocessor. They consist of several basic instructions, called the kernel of microprocessor. These instructions of the kernel can be carried out by a sequential machine, therefore, we can use a checking experiment to verify the kernel.

A. The kernel of microprocessor

Definition 1. Kernel instruction set - A small subset of instructions of a microprocessor which can be used for constituting the write and read register sequences.

Definition 2. Register set - All internal registers of a microprocessor from the view of the architecture or programming.

Definition 3. Kernel state - The register state, i.e. certain set of data values of the registers.

Definition 4. Kernel input - The write sequence for writing a set of data into the registers, or the read sequence for reading out the contents of the registers.

Definition 5. Kernel output - The set of data values of the registers which are read out by the read sequence.

B. Kernel instruction set

There exists many choices for the kernel instruction set. In order to keep the kernel small, the following requirements should be satisfied.

1. The number of instructions in the kernel instruction set should be small.

2. Functions of each kernel instruction should be as simple as possible. For instance, a kernel instruction contains mainly transfer type of operations, or small number of RTL operations.

3. In order to simplify addressing, the priority order of choosing the addressing mode of an instruction is as follows.

. For write register instructions: Immediate, Direct, Indirect.

. For read register instructions: Direct, Indirect.

For the existing off-the-shelf microprocessors, most registers can be written into or read from directly. These registers are called direct access registers. Others are indirect access registers which can be accessed through the direct access registers in certain order by using transfer instruction among registers.

C. Kernel state

In order to simplify the testing, during the checking experiment, we only use a few states for the good kernel, i.e. we define several sets of data values for the registers. Therefore, we should choose the data values (test data) such that they can cover as many faults as possible.

V. TEST DATA

We use the checking experiment to verify the kernel of a microprocessor. The main task is to decide how many states of the kernel and what test data we use.

Abraham and Parker [5] use the k-out-of-m codes for their "register read test" procedure, where m is the width of a code word (i.e. the length of register), and k is the number of 1's in the code word. As we will see, this type of code is powerful since it can be used as test data to cover most control faults by using fewer data. We will use the k-out-of-m codes for verifying the write and read sequences as well as for testing control faults. The k-out-of-m codes can also detect stuck-at type faults, but do not guarantee to cover all data processing faults.

To simplify the testing, we will only use transfer operations of the kernel instructions in write and read sequences. Therefore, we only need to consider the requirements of Theorems 1 and 2.

A. QTT1, QTA1 and QTT2

The k-out-of-m codes used as test data can satisfy requirements QTT1, QTA1 and QTT2 ($V_i \neq V_j$ and $V_i \; I \; V_j \neq V_i$). This is because in the k-out-of-m codes, all code words are distinct and the AND(OR) operation of any two code words will decrease (increase) the number of 1's in the code word, thereby the new code generated is different from both original code words.

B. QTA2

The requirement QTA2, $f'_A \neq VS$, is for detecting fault 'T/A'. Here D and D' are the same register. During the checking experiment for the kernel, there exists an input leaving the kernel state unchanged, i.e., the transfer operation 1 (in write sequence) keeps VD (=VD') unchanged. Therefore $f'_A \neq VS$ becomes $f'_A \neq VS = VD = VD'$ which belongs to QTA3.

C. QTA3

The requirement QTA3, $f'_A(VS'_1, VS'_2, \ldots) \neq VD'$, is for detecting an extra operation. We list the restrictions of operands (test data) for detecting operation of class A in Table 2.

Extra Operations	Restrictions of Operands
BIT SET	①
BIT RESET	①
BIT COMPLEMENT	No
INCREMENT	No
DECREMENT	No
DECIMAL ADJUST	③
SHIFT	$\neq \underline{0}$ (all 0s), $\underline{1}$ (all 1s)
ROTATE	$\neq \underline{0}, \underline{1}$
COMPLEMENT	No
NEGATE	No
CLEAR	$\neq \underline{0}$
ADDITION	$\neq \underline{0}$
ADDITION WITH CARRY	$\neq \underline{0}, -1$
SUBTRACTION	$\neq \underline{0}$
SUBTRACTION WITH BORROW	$\neq \underline{0}, -1$
AND	k-out-of-m codes
OR	k-out-of-m codes
XOR	k-out-of-m codes
EXTEND SIGN	$\neq \underline{0}, \underline{1}$
ADDITION, $D \leftarrow S_1 + S_2$	the least significant bit LSB =1
ADDITION, $D \leftarrow F + S_1 + S_2$	$\neq \underline{0}$; $VS'_1 + VS'_2 \neq 0$
ADDITION, $D \leftarrow S_1 + S_2 + S_3$	④
MULTIPLY	$\neq \underline{0}, 1$
DIVIDE	$\neq \underline{0}, 1$
BIT TEST	②
COMPARE	②
Modifying flags	⑤

Table 2. Restrictions of operands for detecting faulty arithmetic or logical operations

① If we use two tests in which the source operands are complements of each other, then one of the tests can detect the faulty operation.

② These operations only set or reset the flags. We use two tests with the identical source operands and two sets of flags which are complements to each other. The faulty will change one of the flags.

③ The execution of DECIMAL ADJUST (to add certain values) depends on the value of source operand and the flags. We can use either method in case 1 or either method in case 2 to detect the operation.

④ Operation $D \leftarrow S_1 + S_2 + S_3$ is only used for memory address addition. Here D means external address bus. In this case, the unexpected operation does not affect the write and read registers. Hence it needs not to be considered for verifying the kernel.

⑤ During the checking experiment for the kernel, if we have considered the main operations in arithmetic and logical instructions as the unexpected operation, then we need not consider modifying flags which are auxiliary operations.

For other operations, the restrictions are obvious. For example, ADDITION WITH CARRY, $D' \leftarrow D' + S' + CARRY$, if CARRY = 0 with the restriction $VS' \neq 0$ or CARRY' = 1 with the restriction is $VS' \neq -1$, then $VD'^* \neq VD'$. Since we use k-out-of-m codes as operands, these restrictions can be satisfied. For operation $D' \leftarrow D' + S_1' + S_2'$, since the negative value of any k-out-of-m code word will not be any k-out-of-m code, i.e. $VS_1' + VS_2' \neq 0$; so $VD' + VS_1' + VS_2' \neq VD'$. For operation $D' \leftarrow S_1' + S_2'$, we can divide the k-out-of-m codes into two groups with different LSB (Least Significant Bit). These two groups complement each other. The group with the restriction LSB = 1 will be used for testing.

For operation ROTATE, the restriction is for the case of the odd number of shifted bits. Otherwise, we need other restriction (e.g. using subset of k-out-of-m codes as operands).

D. QTA4

The checking experiment does not guarantee requirement QTA4, VS L $F'_A \neq$ VS. For example, for the normal transfer operation T, D←S, with VS = 111000, VD = 011100 (using 3-out-of-6 codes), if there exists a fault $\sigma T/TLA'$ and the extra operation A' is SHIFT LEFT, D←SHL D, then f'_A = 111000. If L is an AND function, then VS L f'_A = VS. Thus QTA4 cannot be satisfied. Therefore, we need another test procedure to remedy this. The remedy is to change the value of S in the operation D←S to $\underline{1}$ or $\underline{0}$ depending upon L being AND or OR respectively. From the above example, we see that if L is an AND function, let VS = $\underline{1}$ then QTA4 becomes $f'_A \neq \underline{1}$ which can be satisfied. Similarly, if L is an OR function, let VS = $\underline{0}$ and QTA4 changes to $f'_A \neq \underline{0}$.

Now let us check this remedy method for all operations of class A. Note that we only change the value of S in operation D←S, and D' in operation A' can be substituted by D.

(i) For class A1, D←f'_A(D). If we use k-out-of-m codes, the new requirement QTA4', $f'_A \neq \underline{1}$ ($\underline{0}$), depending on L, can be satisfied except the operation CLEAR with L being OR. But in this case, the result of f'_A is always $\underline{0}$ which does not affect the operation D←S.

(ii) For class A_2, D←f'_A(D,S'). No matter S' is the same register as S or not, QTA4' is true except the following case: when S' and S are the same register, then for QTA4', $f'_A \neq 1$ (f'_A is operation OR and L is AND) and $f'_A \neq 0$ (f'_A is operation AND and L is OR) cannot be met. But in this case, if S' and S are the same, VS$L f'_A$ is always the same as VS, the fault does not affect D←S, i.e. VS$\wedge$(VD$\vee$VS) = VS and VS$\vee$(VD$\wedge$VS) = VS for any operands D and S.

(iii) For the EXTEND SIGN operation, $D \leftarrow f'_A(S')$, the result of f'_A is $\underline{0}$ or $\underline{1}$. Similarly with case (i), either QTA4 can be met or the fault does not affect operation T.

(iv) For classes A3 and AF, QTA4 can be satisfied. For the same reason as QTA3, we need to consider neither modifying flags operation nor $D \leftarrow S_1 + S_2 + S_3$.

(v) For MULTIPLY and DIVIDE, since the execution period of these operations generally is longer than that of transfer operations, fault $\sigma f/fLf'$ cannot exist and we do not consider QTA4.

In summary, during the checking experiment we only need three sets of test data for internal registers. Let n be the number of the internal registers. m be the length of registers. Suppose that m is even. Let $\underline{V}$ denote a complete set of k-out-of-m code words. Usually, $k = \frac{m}{2}$. $|\underline{V}| = \binom{m}{m/2}$ will be the maximum number of distinct codes. We divide them into two groups, $\underline{V}_0$ and $\underline{V}_1$ with different LSB. These two groups are complementary to each other. Note that for most microprocessors, m is even, and $|\underline{V}| > 2n$.

Let $\{\alpha\} = \{\alpha_1, \alpha_2, \ldots, \alpha_n\}$, $\{\overline{\alpha}\} = \{\overline{\alpha}_1, \overline{\alpha}_2, \ldots, \overline{\alpha}_n\}$, $\{\alpha\}$ and $\{\overline{\alpha}\}$ belongs to $\underline{V}_0$ and $\underline{V}_1$ respectively. We now construct four sets of data as follows.

Flag register	Other registers
α_1	$\alpha_2, \alpha_3, \ldots, \alpha_n$
$\overline{\alpha}_1$	$\overline{\alpha}_2, \overline{\alpha}_3, \ldots, \overline{\alpha}_n$
α_1	$\overline{\alpha}_2, \overline{\alpha}_3, \ldots, \overline{\alpha}_n$
$\overline{\alpha}_1$	$\alpha_2, \alpha_3, \ldots, \alpha_n$

In order to satisfy requirement QTA3, we can choose any three sets of data as the initial values of registers (test data). In fact, one needs a few more data as external bus inputs during testing. Therefore, the final number of code

words in $\{\alpha\}$ and $\{\bar{\alpha}\}$ is larger than n.

It should be pointed out that the program counter (PC) is easy to test. We can put a direct addressing branch instruction at the end of a write sequence. This instruction stores a particular value into PC, then the content of PC is checked at the beginning of the read sequence by observing the address bus.

VI. VERIFYING WRITE AND READ SEQUENCES

Now we will derive the checking sequence for the kernel. As we mentioned before, we only use three sets of test data for the kernel, namely a, b and c. First of all, just like a sequential machine we have to obtain a flow table of the kernel. We consider two cases.

Case 1. For a microprocessor without indirect access registers, we obtain the following flow table.

	W_a	W_b	W_c	R
A	A (1)	B (4)	C (7)	A,a (10)
B	A (2)	B (5)	C (8)	B,b (11)
C	A (3)	B (6)	C (9)	C,c (12)

Case 2. For a microprocessor with indirect access registers, we obtain the following table.

	W_a	W_b	W_c	R
A	A (1)	B (7)	C (13)	A*,a (19)
B	A (2)	B (8)	C (14)	B*,b (20)
C	A (3)	B (9)	C (15)	C*,c (21)
A*	A (4)	B (10)	C (16)	A*,a* (22)
B*	A (5)	B (11)	C (17)	B*,b* (23)
C*	A (6)	B (12)	C (18)	C*,c* (24)

where W_a, W_b, W_c- write sequences for writing a, b and c respectively.

R - read sequence.

A, B, C, A*, B*, C* - Kernel states. A, B and C are the states after applying the corresponding write sequences W_a, W_b, W_c. A*, B* and C* are the states after applying read sequence R.

a, b, c, a*, b*, c* - Kernel output sequences produced by read sequence R for states A, B, C, A*, B*, C* respectively.

ⓘ denote the state transition i.

Since we only use three sets of test data, the number of states of the kernel is constant. It means that the above flow tables are independent of microprocessors. Therefore, we can easily obtain the checking sequences with the same form.

There are three requirements to derive a checking sequence [12]

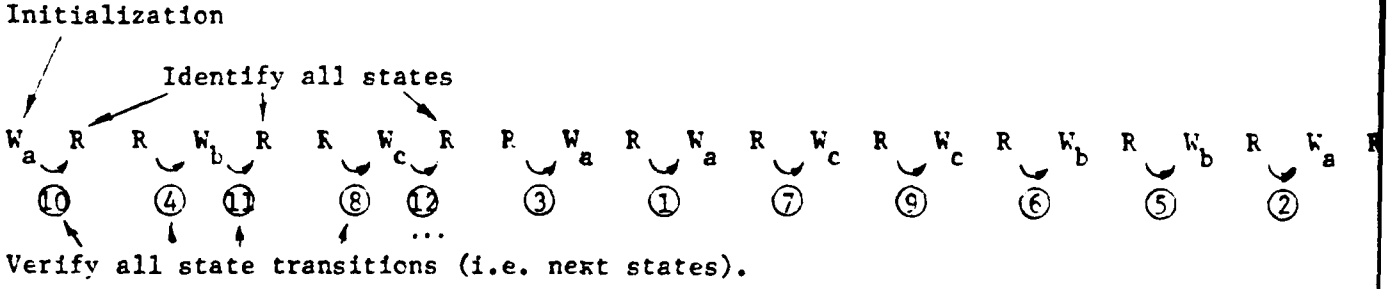
1. Initialization of the machine (kernel) state using sychronizing sequence or homing sequence.

2. Identify all machine states using distinguishing sequence.

3. Verify each transition using distinguishing sequence.

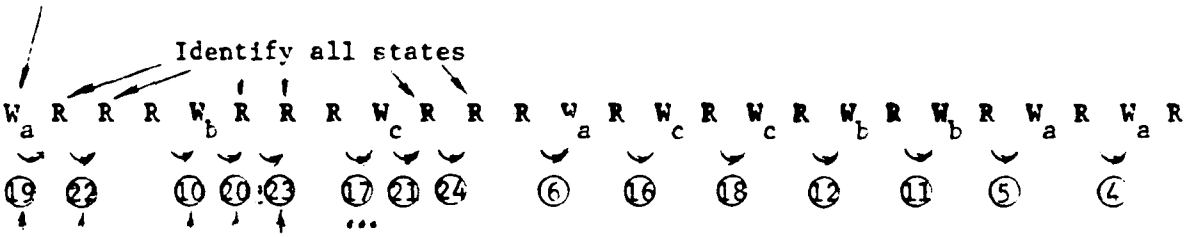
For our kernel, there exists sychronizing sequences W_a or W_b or W_c and distinguishing sequence R. Therefore, we can easily derive checking sequences as follows.

Checking Sequence 1 (for case 1)

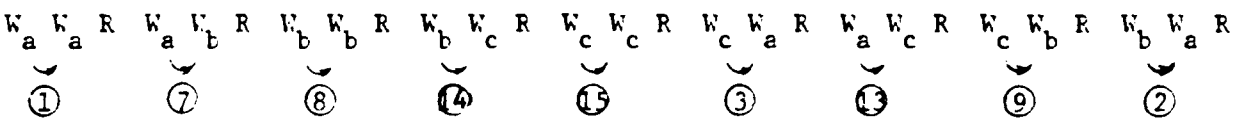


Checking sequence 2 (for case 2)

Initialization



Verify all state transitions (i.e. next states)



Finally, we can obtain two test procedures for verifying the write and read sequences as follows.

Procedure 1: Checking experiment

1. If a microprocessor does not have indirect access registers, we use checking sequence 1.
2. If a microprocessor has indirect access registers, we use checking sequence 2.

Procedure 2: Remedy testing

For each of the three sets of register values a, b and c, do the following for each register.

1. Initialization of all registers.
2. Write 1 or 0 into the given register depending on the circuit implementation.
3. Read the given register.

Therefore, from the previous discussion in this section, we obtain the following theorem.

Theorem 5. Procedures 1 and 2 can verify write and read sequences, and after that the registers of a microprocessor can be initialized to any values.

VII. TESTING CONTROL FAULTS

During the verification of the correctness of write and read sequences, we are only concerned with certain transfer operations (not all RTL operations) in the kernel instructions. Therefore, when we test instruction decoding and other control faults, we need to test all instructions included in the kernel. Note that obviously, register decoding faults can be detected by verifying write and read sequences using the k-out-of m codes.

Procedure 3. Testing instruction decoding and other control faults

For each instruction, do the following test,

1. Initialize register state using any particular initial values. (test data)
2. Execute the instruction-under-test.
3. Read register state.

Note that we should first try to use three sets of data values a, b and c at Step 1.

Generally we need several tests for each instruction to detect the instruction decoding and other control faults. Obviously, the lower bound of the number of tests using Procedure 3 for each instruction is two. This is because any kind of microprocessors has several pairs of conditional branch instructions based on two different values for the same condition sourse. Therefore, when any instruction is under test, in order to detect an unexpected branch instruction due to a fault, we need at least two test patterns.

The upper bound on the test for each instruction is dependent upon the microprocessor-under-test. We can roughly estimat the order of tests for detecting instruction decoding faults. We consider n_I instructions to be tested, assume that each instruction corresponds to an operation, class T or Class A, used for distinguishing instructions from each other. Let n_{IT} and n_{IA} denote the number of instructions which have operations class T and

class A respectively, i.e. $n_I = n_{IT} + n_{IA}$.

A. Testing instruction class T

1. The case of Theorem 1

Since for some condition branch instructions, their operations belong to class T, and they are condition transfer operations. In order to detect this unexpected operation T', two tests, in which test data are complement to each other, are sufficient.

2. The case of Theorem 2

First of all, we use three sets of initial values a, b and c for satisfying the requirements QTA1 and QTA3. In order to satisfy QTA2 and QTA4, we can modify a, b and c separately as new test data. As we have discussed in Section V, if the instruction-under-test has a transfer operation D←S, we can change VS in original data a, b and c to VD, then QTA2 becomes QTA3 which can be satisfied. Similarly, we can change VS to 1 (or 0) to satisfy QTA4. Here we need nine tests altogether. Thus, the order of the number of tests for testing instruction class T is $O(n_{IT})$.

B. Testing instruction class A

1. The case of Theorem 3

Test data a, b and c can cover QAT1 and QAT3. Similarly, in order to satisfy QAT2, QAT4, QAT5 and QAT6, we can modify a, b and c in turn. First, we change VS' and VD' to VD for covering QAT2 and QAT4 respectively. These changes are done for each register, so it needs 3n tests, where n is the number of registers. Then we change VS' to a particular value for covering QAT5 and QAT6 separately. It needs 6n tests. So the total number of tests for each instruction in this class will be 9n+3.

2. The case of Theorem 4

We need three tests using a, b and c for covering QAA1 and QAA2.

Then we attempt to find five particular tests to satisfy QAA3 through QAA7 for each unexpected operation A'. So the number of tests for each instruction in this class will be $3 + 5\ (n_{IA} - 1) = 5\ n_{IA} - 2$.

Therefore, the order of the number of tests for testing instruction class A is $0\ (n \cdot n_{IA} + n_{IA}^2)$. The order of the number of tests for testing instruction decoding and other control faults is $0\ (n_{IT} + n.n_{IA} + n_{IA}^2)$. Note that using Thatte and Abraham's approach [3], the order of the number of tests for testing instruction decoding faults is $0\ (n_I^2)$.

VIII. CONCLUSION

Any deterministic functional testing approach for microprocessors always involves the initialization and the reading of internal registers in each test, i.e. write and read sequences. If we devide the testing into two steps and guarantee the write and read sequences' correctness first, then the complexity of test generation will be reduced.

Since control faults possibly lead to the partial execution of an instruction or changing the execution sequence, we assert that it is reasonable to define a control fault model at the RTL level instead of the instruction level.

For test generation, usually one derives a test for a given fault. But if we find a test to cover as many faults as possible, then test generation will be more efficient. It seems that the k-out-of-m codes for test data are quite powerful.

The function of write and read sequences can be modeled as a small sequential machine which is almost independent of microprocessors. Therefore, we can easily use the checking experiment to verify the sequential machine.

Further work includes the enumeration of the control faults at the RTL level for the generation of tests to cover all possible faults.

In addition, for the data processing part of a microprocessor, it is easy to handle storage faults and transfer faults. For data manipulation faults, it seems that the best way is to find a set of test patterns to exercise each operation based on the analysis of logic function of the operation, or simply use random data (locally).

Combining the simplified functional testing method with the signature analysis technique may be a good approach for implementing the Built-In Self Test (BIST) of microprocessors.

ACKNOWLEDGEMENT

The authors sincerly thank Dr. K. K. Saluja and Dr. P. Thevenod-Fosse for their helpful comments and suggestions on this paper.

REFERENCES

[1] A.C.L. Chiang and R. McCaskill, "Two New Approaches Simplify Testing of Microprocessors", Electronics, 22 Jan. 1976, p. 100.

[2] S.M. Thatte and J.A. Abraham, "Methodology for Functional Level Testing of Microprocessors", 8th International Symposium on Fault-Tolerant Computing, Toulouse, France, June 1978, pp. 90-95.

[3] S.M. Thatte and J.A. Abraham, "Testing Generation for Microprocessors", IEEE Trans. on Computers, C-29, No. 6, June 1980, pp. 429-441.

[4] J.A. Abraham and S.M. Thatte, "Fault Coverage of Testing Program for a Microprocessor", 1979 Test Conference, Oct. 1979, pp. 18-22.

[5] J.A. Abraham and K.P. Parker, "Practical Microprocessor Testing: Open and Closed Loop Approaches", IEEE Compcon, Spring 1981, pp. 308-311.

[6] Y. Min and S.Y.H. Su, "Testing Functional Faults in VLSI", 19th Design Automation Conference, Los Vegas, Nevada, 1982, pp. 384-392.

[7] K.K. Saluja, L. Shen and S.Y.H. Su, "A Simplified Algorithm for Testing Microprocessors", 1983 Test Conference, Oct. 1983, pp. 668-675.

[8] D. Brahme and J.A. Abraham, "Functional Testing of Microprocessors", IEEE Trans. on Computers, C-33, No. 6, June 1984, pp. 475-485.

[9] C. Robach and G. Saucier, "Microprocessor Functional Testing", 1980 Test Conference, Nov. 1980, pp. 433-443.

[10] M.A. Annaratone and M.G. Sami, "An approach to Functional Testing of Microprocessors", 12th International Symposium on Fault-Tolerant Computing, Santa Monica, CA, June 1982, pp. 158-164.

[11] B. Courtois, "Functional Testing of the Control Section of Integrated Processing Units", RR No. 203, IMAG, University of Grenoble, France, May 1980.

[12] A.D. Friedman and D. Menon, Fault Detection in Digital Circuits, Prentice-Hall, NJ, 1971.

PART II

FUNCTIONAL TEST GENERATION OF DIGITAL LSI/VLSI SYSTEMS USING MACHINE SYMBOLIC EXECUTION TECHNIQUE*

by

Tonysheng Lin and Stephen Y.H. Su

Research Group on Design Automation
and Fault-Tolerant Computing
Thomas J. Watson School of Engineering,
Applied Science and Technology
State University of New York
Binghamton, NY 13901

ABSTRACT

This paper presents a new algorithm for functional test generation of digital LSI/VLSI systems. First, a register-transfer (RT)-level fault model is developed based on a well-defined register-transfer-language (RTL). The analysis and collapsing of faults in RT-level fault model were performed. Then, the RT-level symbolic execution technique is employed. The major problems encountered are defined, analyzed and solved. Finally, an explicit algorithmic test generation algorithm is developed. This practical algorithm applies software skills in hardware testing. It is easy to be automated and hence provides a promising solution for future testing problems of digital LSI/VLSI systems.

1. INTRODUCTION

Due to the rapidly increasing complexity of modern LSI/VLSI devices, functional testing has received more attention than ever [1-3]. The high complexity of VLSI makes conventional gate-level testing very difficult and expensive to perform. Techniques for design-for-testability (DFT) or built-in-self-test (BIST) consider the testing problem during the design stage of digital devices with the objective of reducing the test complexity. However, these approaches are not applicable to the existing off-the-shelf components (e.g., Z-80, 8080). Function-level testing is another promising solution to solve these problems. Functional testing can be used to assure proper operations of a system with off-the-shelf components and to "test" (and hence improve) a digital system even in its design phase.

Functional-testing uses a representation of a digital system higher than the gate-level testing. In functional testing, functional faults with respect to the functional specification (e.g., addition operation in a processor) are tested instead of signal faults (e.g., stuck-at-0) at the inputs and the output of a logic gate or interconnections among gates in gate-level testing. The purpose of functional testing is to validate correct functional operations of digital systems according to their specifications. Using functional testing techniques, one cannot only reduce the test generation complexity but also obtain a test set for testing the digital devices with the same functions but different circuit design/implementation (e.g., parallel adder vs. serial adder). Especially for the users of LSI/VLSI chips, they have little other alternative but functional testing since the chips' design/implementation details are usually considered proprietary.

Several functional testing techniques have been proposed today. Su and Lin [3] recently overviewed most of the major techniques in literature. Shen and Su [4] discussed the major issues involved in functional testing of microprocessors. Brahme and Abraham [5] proposed a new fault model for the control and instruction decoding faults to increase the practicality of their previous approach [15]. Based on the observations obtained in [3], most, if not all, of the existing techniques need further development. There are still a lot of open problems which need to be solved.

In this paper, a new algorithm to systematically perform functional test generation for digital LSI/VLSI systems using machine symbolic execution technique is presented. The digital system under test is described by the popular register transfer language (RTL). This technique is based on two major foundations. First, after a standard syntax of a register transfer statement is defined, a comprehensive RT-level fault model is set up. All types of faults covered by the fault model are analyzed and the number of fault cases is reduced. More practical faults than other techniques are included in this fault model. Secondly, based on the RT-level fault model derived, the technique of symbolic execution which is intensively developed in high-level programming languages is employed in RT-level test generation. Symbolic execution is a kind of program execution technique which manipulates symbolic variables instead of variable values during program execution. Since the syntax structure and operational complexity of RT-statements are much simpler than those of higher level languages, the problems of symbolic execution encountered in RT-level are less complex. Basically, the symbolic execution is performed on both fault-free and fault-injected machines. By comparing the symbolic results and path constraints obtained from fault-free and fault-injected machines, the input test patterns to distinguish "bad" machines from "good" ones can hence be derived.

The formal syntax definition of the standard register transfer statement (RT-statement) is given in Section 2. Section 3 presents the analysis and collapsing of the RT-level faults. The basic pro-

* This work is supported by the United States Army Communication Electronics Command under Research Contract No. DAAB 07-82-K-J056.

blems of machine-level symbolic execution and their solutions are described in Section 4. Section 5 outlines the steps in the overall test generation algorithm and explains the key ideas within each step. Finally, a brief discussion along with concluding remarks is given in Section 6.

2. THE REGISTER TRANSFER DESCRIPTION

The register transfer (RT) description is a powerful, and hence, popular tool for describing digital systems. The RT-language introduced here uses the commonly adopted syntax notations. Its design is mainly intended for the use of functional representation and functional level test generation of the digital system under test. With a little modification, it can be extended for use in other related applications such as formal machine specifications in computer-aided-design (CAD).

The register-transfer level description of a digital system is complete with two distinct descriptive parts: the non-executable part and the executable part. The non-executable part consists of a set of declaration statements. There are three kinds of basic declaration statements in this part: INPUT, OUTPUT, and INTERNAL. INPUT declares the input registers of the system. The OUTPUT and INTERNAL declare the output registers and the internal registers respectively. Following the non-executable part is the executable part. This part is composed of register transfer statements with syntax to be discussed in the next few paragraphs. At the end of the executable part, we use an "END" to indicate the termination of the overall RTL description of the system. To enable easy reading and understanding of the RT-statements, commentary statements enclosing at both ends with "%" symbol may be inserted anywhere in the description.

A number of operands are defined here representing arithmetic operations, logical operations, shifts, field extractions, and bit string concatenation. The domain and range of these operands are bit-strings. The default value for bit string is 2's-complement for arithmetic operations and unsigned for logical operations. Both decimal and binary integer constants may be used in the statements. Each operator has well-defined rules for the resultant bit-length as a function of the sizes of input operands. In addition, some operations include implicit sign- or zero-extension of shorter operands to match a longer one. Figure 1 lists those operators defined here in group of their natures. Where, @ is the concatenation operator and .. is the field extraction operator.

```
Unary-operators:: = unary-  NOT | INC | DEC
adding-operators:: = +  -
logic-operators:: = AND  OR  XOR | NOT
relational-operators:: =  =  <>
representational
  operators:: =  shf | shr | @ | ..
```

Figure 1 Operators defined in RT-statements

A typical RT-statement syntax is given below:

$$k: (t,c)\ R_d \leftarrow f(R_{s1},\ldots,R_{si}),\rightarrow n \qquad i\in\{1,2\}$$

where k, t, c, R_d, f, R_{si}, →n, and ← are called RT-components. The meaning of each is briefly described below:

- k: is a positive integer representing the label of an RT-statement.
- t: is a one-bit value timing flag.
- c: is the condition expression with relational operator specifying the condition for performing the register transfer operation.
- R_d: denotes the destination register of the RT-statement.
- R_{si}: is the i-th source register.
- f: stands for an ALU operator operating on the content(s) of source register(s).
- ←: represents the transfer of the result of RT-operation to the destination register.
- →n: is the label of RT-statement to be jumped to after current RT-statement is finished and t and c are true if present.

The operands in RT-expressions may be regular registers with explicit bit-length, constant registers with or without explicit bit-length, or macro registers with explicit bit length. A macro register is a group of registers obtained by concatenating two or more registers. Note that memory reference can also be directly expressed in an RT-statement. All memory references are represented by a memory register. The above typical RTL statement includes the following types of RT-statements as special cases:

- Pure transfer statement: k: $R_d \leftarrow R_s$,→n
- Data operation statement: k: $R_d \leftarrow f(R_{si})$,→n
- Conditional branch statement: k: (c),→n
- Constant transfer statement: k: $R_d \leftarrow$#C,→n

Figure 2 shows the RT-description for a hypothetical machine called SIMPLE-CALCULATOR. After the first three lines of declaration, the RT-description of the SIMPLE-CALCULATOR starts the instruction fetch cycle followed by the instruction decoding cycle and then the instruction execution cycle. More discussion of the defined RT-language can be found in [6].

```
%SIMPLE CALCULATOR%
%DECLARATION PART%
INPUT: START(1), DBS(1), DB(8), OP(3)
OUTPUT: AS(1), A(8), O(8), F(1)
INTERNAL: BS(1), B(8), E(3), SC(3), QS(1), S(1)

%PROGRAM PART%
 0: S ← START, → 1
 1: (S=0), → 0
 2: E ← OP, → 3
   %INSTRUCTION DECODING CHAIN%
 3: (E=001B), → 11 %ADD%
 4: (E=010B), → 12 %SUBTRACT%
 5: (E=011B), → 13 %MULTIPLY%
 6: (E=100B), → 21 %LOAD A%
 7: (E=101B), → 23 %LOAD B%
 8: (E=110B), → 25 %LOAD O%
 9: (E=111B), → 27 %LOAD SC%
10: S ← 0, → 0 %OTHERWISE%
11: F @ A ← A + B, → 10
12: F @ A ← A - B, → 10
13: AS ← BS XOR QS, → 14
14: A ← 0, → 15
15: F ← 0, → 16
16: (Q(7)=1) F @ A ← A + B, → 17
17: F @ A @ Q ← SHR F @ A @ Q, → 18
```

```
18: SC ← SC-1, → 19
19: (SC <> 0), → 16
20: S ← 0, → 0
21: A ← DB, → 22
22: AS ← DBS, → 10
23: B ← DB, → 24
24: BS ← DBS, → 10
25: Q ← DB, → 26
26: QS ← DBS, → 10
27: SC ← DB(1..3), → 10
END
```

Figure 2 An example of RT-description

3. THE RT-LEVEL FAULT MODEL AND FAULT COLLAPSING

Based on the typical RTL statement given in the last section, nine types of RT-level faults can be derived. The functional effect of these faults can be analyzed. The results are described in several lemmas. Based on these lemmas, analysis for RT-level fault collapsing is conducted and several important theorems are derived. For brevity only the major lemmas and theorems will be presented in this section. Detailed discussions of the overall development can be found in [6].

Definition 1. After a digital system is described by a set of RT-statements, the overall behavior of the system is determined by the resultant functions of the associated RT-statements. Each basic component in an RT-statement is called a register-transfer (RT)-component.

Definition 2. A digital system is said to be fault-free if it operates correctly with respect to its functional specifications. Otherwise, it is said to be faulty due to some register-transfer level fault occurring in the basic function component of the system. Faults which are considered in the system may be classified according to their fault effect on certain RT-component as fault type. The different fault values under each occurrence of fault type is called fault case.

Definition 3. When an RT-component F becomes faulty with fault type F', we denote it by F/F'.

Based on the typical RTL statement, RT-level faults can be classified into the following nine types.

- k/k' : label fault
- t/t' : timing fault
- c/c' : condition fault
- (R)/(R)' : data storage fault
- ←/←' : data transfer fault
- R/R' : register decoding fault
- f/f' : operator decoding fault
- (f)/(f)' : operator execution fault
- →n/→n' : jump fault

The following lemmas are established:

Lemma 1. Fault k/k' means that the label of an RT-statement changes from k to k'. If stuck-at-fault is assumed, then possible faulty situations are:

(i) $k' \equiv \phi$, label k disappears.
(ii) $k' \equiv k_1$, k_1 is another faulty label with one bit different from k within the system's RT-address space.
(iii) $k' \equiv k_2$, k_2 is a non-existent label with one bit different from k within the system's RT-address space.

where, single-bit stuck-at fault is assumed in (ii) and (iii).

Proof:

(i) If k/k' occurs, one of the possible faulty cases is that the addressing mechanism for k is totally masked. In this case, no k, therefore, will be found by the system's execution mechanism.

(ii) (iii) Suppose A_s is the maximum valid address in the RT-address space of current digital system and B is the bit length of label k. When k/k' occurs, there are B faulty possibilities k_i', $1 \leq i \leq B$, with only one-bit different from the correct bit value of k. For those k_i' with value $k_i' \leq A_s$, they belong to the fault type $k' \equiv k_1$. For those k_i' with the value $k_i' > A_s$, they belong to the fault type $k' \equiv k_2$.

To save space, following lemmas are stated without proof. The detailed proofs can be found in [6].

Lemma 2. The jump section of an RT-statement may become faulty. We use the notation →n/→n' to represent that such type of fault occurs. If bridging fault and stuck-at-fault are assumed in the jump section, then the possible faulty situations are:

(i) $n' \equiv \phi$, label n disappears.
(ii) $n' \equiv n_1$, n_1 is another faulty label with one bit different from n within the system's RT-address space.
(iii) $n' \equiv n_2$, n_2 is a non-existent label with one bit different from n within the system's RT-address space.
(iv) $n' \equiv n + n_1$, both n and n_1 are simultaneously activated. n_1 is another label defined in (ii).

where, single-bit stuck-at-fault is assumed in (ii) and (iii).

Lemma 3. The timing signal of an RT-statement may be faulty and is represented by t/t'. When t/t' occurs, the corresponding RT-statement will not be executed.

Lemma 4. The condition control mechanism of an RT-statement may be faulty and is represented by c/c'. When c/c' occurs, the reverse condition instead of the normal condition is true. Under this case, the corresponding RT-statement will not be executed.

Lemma 5. The content of a register may be faulty and is represented by (R)/(R)'. We assume that stuck-at and bridging faults are considered in this case and B= R is the bit length of register R.

(i) If R is a regular register which can be read or written, then every bit in R may either be stuck-at-0 or stuck-at-1, and every bit-pair combination in R may also be bridged together. The total number of multiple stuck-at-faults is 3^B-1 and the total number of bridging faults is B x (B-1)/2.

(ii) If R is a constant register, then the total number of fault cases is B by considering only single-bit stuck-at faults.

Lemma 6. The content in a register transfer path may be faulty and is represented by ←/←'. If stuck-at fault and bridging fault are considered, then every bit in the path may be stuck-at-0 or stuck-at-1 and every bit-pair combination in the path may also be bridged together. The total number of

multiple stuck-at faults is 3^B-1 and the total number of bridging fault is B x (B-1)/2, where B is the bit-width of the transfer path.

Lemma 7. The decoding of a register may become faulty and is represented by R/R'. Assume at most one register will be selected at a time, then the possible fault cases are:
(i) R'≡ε, no register is chosen.
(ii) R'≡R_2, R_2 is another register with similar register characteristics in the system.

Lemma 8. The selection of an operator in the Arithmetic Logic Unit (ALU) may be erroneously performed and is denoted by f/f'. Assume at most one operator will be selected at a time. The possible faulty situations are:
(i) f'≡ε, no operator is chosen.
(ii) f'≡f_2, f_2 is another valid ALU operator different from f.

Lemma 9. The execution of an operator in ALU may be faulty and is denoted by (f)/(f)'. Due to the nature of the faulty effect of (f)/(f)', this type of fault is difficult for modeling in register transfer level and may only be attacked at the circuit/gate-level or implementation-dependent level.

Based on the above Lemmas, RT-level fault collapsing analysis similar to that of gate-level stuck-at-faults can be performed. The result is described in the following theorems.

Definition 4. Two RT-level faults F_1 and F_2 in a digital system are said to be functionally equivalent if and only if their faulty results observed in the RT-level description of the system are identical.

Theorem 1. In an RT-description, the k/k' type of fault is covered by (a subset of) the →n/→n' type of fault by Lemma 1 and Lemma 2. That is:
(i) Both k/k'≡ε and k/k'≡k_2 are functionally equivalent to →n/→n'= →k/→n_2.
(ii) k/k'≡k_2 is functionally equivalent to the combination of →n/→n' = →k/→n_2 and →n/→n' = →k'/→(k_2).

Proof: Suppose in the RT-description, there are certain RT-statements $\{S_i\}$ with jump section →k, and there exists an RT-statement S_k with statement label k.
(i.a) When label fault k/k'≡ε occurs, $\{S_i\}$ are not affected. Whereas the label of S_k becomes ε. Whenever any S_i in $\{S_i\}$ is executed, the next RT-statement to be executed is S_k in fault-free case. Now, since k/k'≡ε occurs, there is no RT-statement with label k anymore. S_k is virtually non-existent. Therefore, any S_i in $\{S_i\}$ will actually activate a trap to a non-existent address. In other words, the result of the appearance of k/k'≡ε in S_k turns out to be the fault of →k/→n_2 in each RT-statement of $\{S_i\}$, where n_2 is a non-existent RT label in the RT-address space.
(i.b) When k/k'≡k_2 occurs, $\{S_i\}$ are not affected. Whereas the label of S_k becomes k_2 which is non-existent in valid address space of the RT-description. Whenever any S_i in $\{S_i\}$ is executed, the next RT-statement to be executed is S_k in fault-free case. Now, since k/k'≡k_2 occurs, the original S_k can no longer be addressed within the valid address space. That is, every S_i in $\{S_i\}$ will activate a trap to a non-existent address k. In other words, the result of the appearance of k/k'≡k_2 in S_k turns out to be functionally equivalent to the fault of →k/→n_2 in each S_i of $\{S_i\}$.
(ii) Suppose in the RT-description there are other RT-statements $\{S_j\}$ with jump section →ℓ, and there exists an RT-statement $S_ℓ$ with label ℓ. When k/k'≡k_2≡ℓ occurs, both $\{S_i\}$ and $\{S_j\}$ are not affected. Whereas the label of S_k becomes k_2≡ℓ. Whenever any S_j in $\{S_j\}$ is executed, the next RT-statement to be executed are S_k and $S_ℓ$ instead of $S_ℓ$ alone in the fault-free case. And any S_i in $\{S_i\}$ now will activate a trap to a non-existent address k. In other words, the result of the appearance of k/k'≡k_2≡ℓ in S_k turns out to be functionally equivalent to the combined faults of →k/→n_2 in each S_i of $\{S_i\}$ and →ℓ/→ℓ' =→ℓ/→ (ℓ+k) in each S_j of $\{S_j\}$. The notation "ℓ+k" indicates that the RT-statement with label ℓ and the RT-statement with label k are both executed.

Theorem 2. Assume the internal paths among registers of a digital system are all parallel links. In the RT-description of such a digital system, the (R)/(R)' type of fault defined in Lemma 5 is covered by the →/→' type of fault defined in Lemma 6. After all →/→' type of faults are tested, the (R)/(R)' type of faults are automatically tested.
Proof: If a register is redundant in the system, then either its behavior is transparent to the system specification or it is meaningless to the correct system operation (e.g., design error). For every non-redundant register R_i in an RT-description, there always exists at least a path in the system to transfer the content of R_i out to a destination register, say, R_j. Whenever the content of R_i is moved out, its value is copied onto the transfer path. Therefore, whatever $(R_i)'$ faults will be mapped onto its associated transfer path in the RT-statement. If all →/→' fault cases on those transfer paths connecting all register $\{R_i\}$ in the RT-description are tested, then all fault cases of $(R_i)/(R_i)'$ are automatically tested.

Theorem 3. In an RT-description, fault t/t' is covered by the combination of →n/→n' and c/c' types of faults by Lemma 2, 3 and 4. That is:
(i) When t is present in an RT-statement while c is absent, t/t' is functionally equivalent to →n/→n'≡ n+1, where n is the label of this RT-statement.
(ii) When both t and c are present in an RT-statement t/t' is functionally equivalent to c/c'.
Proof: (i) By the definition of t/t', when t' occurs, the associated RT-statement will not be executed. The next RT statement to be executed is hence the RT-statement next to current statement in the RT-description. This is exactly what happens when →n/→n'≡n+1 occurs.

(ii) In general digital systems, the timing control signal is first applied to activate the associated RT-statement. If t' occurs, the corresponding RT-statement will not be activated. When t is active and fault-free, then c is checked if present. If c is true, then the associated RT-statement will be executed otherwise it will not. However, if c c' occurs, the associated RT-statement will not be executed even if t is valid. Hence, t/t' and c c' both produce the same faulty behavior, and we may conclude that after c/c' is tested, the t/t' type of fault of the associated RT-statement is also automatically tested.

Using the above three theorems as basis, we establish the following two theorems for test generation.

Theorem 4. In an RT-description, not every RT-level fault type defined in Lemma 1 through Lemma 8 need be considered for a complete test set under the defined RT-level fault model. In other words, the number of fault types can be reduced. First, the eight modeled RT-level fault types can be collapsed to five by Theorem 1 through Theorem 3. Second, only one of those RT-components which functionally correspond to the same part of hardware need be considered for fault cases.

Proof: (The proof is lengthy and omitted here.)

Theorem 5. A complete test set for the fault model stated in Lemma 1 through Lemma 8 can be derived if the following five fault types in every RT-statement are considered with each fault case in each fault type being considered only once:

R/R', f/f', +/+', -/-', and c/c'

Proof: The proof is obvious by Theorem 4. Suppose every RT-statement in the RT-description is scanned serially according to a predefined order derived from the RT-description. If any of the above five fault types F_i' occurs in an RT-statement and the tests for F_i' are not yet derived, then its test set will be generated and added to the tests obtained so far. If F_i' is already tested somewhere before current RT-statement, then it is skipped. The complete test set can hence be computed using this procedure repeatedly until every RT-statement is processed in this way.

4. MACHINE SYMBOLIC EXECUTION TECHNIQUE

The symbolic execution is a very useful software engineering technique developed originally for program analysis including test data generation [7,8] and program validation [9,10]. Due to the similarities between software and hardware implementation, most key principles in this powerful technique for software analysis may also be used for hardware testing and verification [11,12]. Su and Hsieh [13] first pointed out the general idea of generating tests for digital systems by symbolically executing the fault-free and fault-injected machines.

Symbolic execution is a process of program execution similar to normal execution except that symbolic values of variables and their operation rules are included in addition to normal ones. It involves assigning expression instead of values to variables while following a program path. An expression represents the computation that would have evolved to compute each variable's value. A symbolic value is an expression of constants and variables whose contents are fixed but unknown during the symbolic execution. During the process of symbolic execution, every internal variable can always be described as an expression of constants and symbolic values of external input variables. When the symbolic execution proceeds, a binary tree, called symbolic execution tree (SET) which shows paths of all possible symbolic execution flows of a program will be developed. If a particular symbolic execution path is followed up to its termination, a set of path constraints and that path's symbolic results will be obtained. Since a symbolic execution takes the symbolic values of all its external input variables as input data, a substantial number of actual input data combinations are included under the path constraints within a single symbolic execution of the program. By a straightforward computation the desired input test patterns which distinguish fault-free from faulty operations may be systematically computed. It is this important feature of symbolic execution that makes it a powerful tool in the test generation algorithm under development in this research.

Although symbolic execution has existed for a long time as a means of determining the semantics of programs, it, however, has been invariably used mostly for programs written only in a high-level language. Little work has been done in the domain of symbolic execution of formal machine description. Oakley [14], however, did a good job in laying the basic theoretical foundation in the investigation of this topic. Indeed, there are two major differences between machine-level symbolic execution (MSE) and high-level-language symbolic execution (HSE). First, the major goal of HSE is to confirm that the programmer has put together his software program statements in a correct manner, whereas MSE is mainly concerned with the correct functioning of the hardware primitives themselves. The second major difference is that due to human factors and the high complexity of software design, HSE is still very difficult and restricted to use in many applications. The MSE, on the other hand, is more amendable to a systematic method because of the relative simplicity of the definition of the hardware description language (e.g., the RT- language defined in Section 2), the simpler way of hardware description and so on. While the above statements are true, MSE still has its own problems. The basic issues in MSE are: equivalence of machine symbolic execution and actual machine execution, variation of bit-width among internal register paths, the representation of symbolic context and path constraints, the problem of loops and termination during symbolic execution, and the simplification of symbolic expressions. A detailed discussion of all of these and their solutions can be found in [6,14] and is omitted here.

The symbolic execution system developed in this research is composed of four major modules and has the system organizational structure as shown in Figure 3.

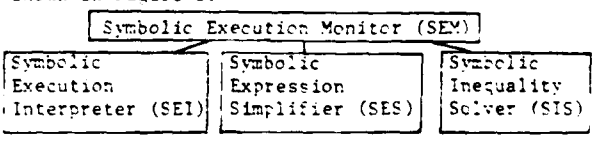


Figure 3 Organization of the symbolic execution system

In Figure 3, SEM is for the overall control of symbolic execution process. SEI interprets the semantics of an RT-statement. SES performs symbolic expression simplification. SIS is called only when the symbolic results of the fault-free and fault-injected paths are obtained.

The whole symbolic execution system constitutes a modular part of the overall test generation algorithm to be discussed in the next section. It acts in a coordinated way with other control mechanism in the test generation algorithm. From the operational standpoint, the internal execution flow of the whole symbolic execution system is depicted in Figure 4. The execution loop between SEI and SES will continue until the last RT-statement on the current selected path is executed. At that time, SIS will be activated to commence execution.

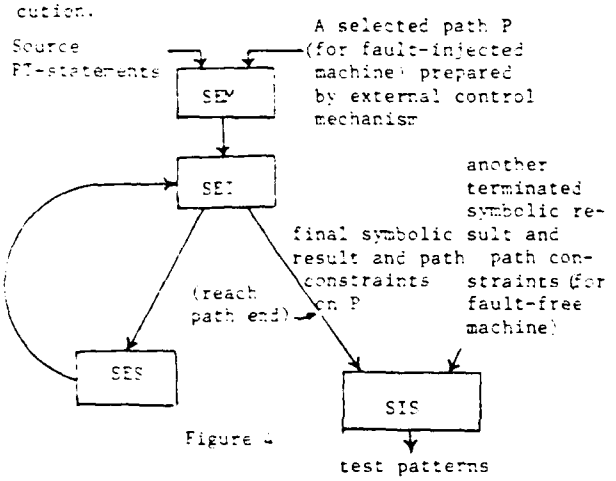


Figure 4

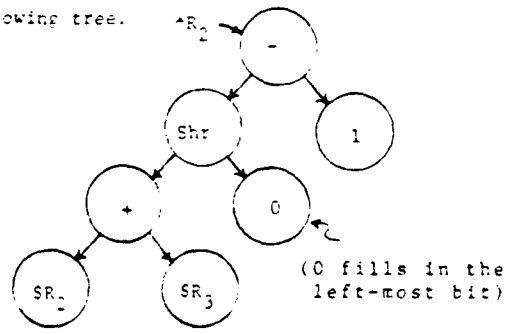
The symbolic result is represented internally as a directed tree. One of the features of tree structure is the easy handling of its growth. As an example, consider the following three RT-statements.

5: $R_1 \leftarrow R_2 - R_3, \rightarrow 6$

6: $R_3 \leftarrow shr\ R_1, \rightarrow 7$

7: $R_2 \leftarrow R_3 - 1, \rightarrow 0$

After statement 7 is executed, the symbolic value of R_2 will be $shr(SR_2 + SR_3) - 1$ which is represented by following tree.



where, SR_2 and SR_3 denote the primary symbolic value of input register R_2 and R_3 respectively. $^*R_2$ represents the current symbolic value of R_2.

5. THE TEST GENERATION ALGORITHM

In this section, the design of an explicitly defined systematic test generation algorithm is presented. The overall test generation algorithm is developed based on the RT-level fault model discussed in Section 3 and the RT-level symbolic execution technique described in Section 4. There are mainly three design considerations behind this functional level test generation algorithm:

1) "Divide and conquer" to partition a big problem into smaller ones.

2) Modularity and Flexibility for steps in the algorithm.

3) Algorithm vs. heuristics for better efficiency.

Before we go into details of the test generation algorithm, several basic assumptions and definitions of common terminologies are in order.

Definition 5. A functional fault is redundant if its appearance in the digital system does not affect the correct functional operations specified for that digital system. Otherwise, it is non-redundant.

Definition 6. A functional fault is itself testable if it is non-redundant and its faulty effect can always be observed at the output port of the system. Otherwise, it is untestable.

Definition 7. A functional fault is detectable by a certain test set if it is observable and by using proper exercising inputs, its faulty effect can be observed at the output port of the system. Otherwise, it is undetectable to that test set.

Definition 8. A functional fault is enumerable if it is non-redundant and can be enumerated within a certain upper limit based on reasonable assumptions. Otherwise, it is non-enumerable.

Assumption 1. In the test generation algorithm we map the physical transfer mechanism such as a bus described in the RT-description into a logical transfer path. This is because test engineers working in a user environment may not know the detailed implementation of the mechanisms for transferring data between functional units, or how they are shared or multiplexed among different RT-statements. Using the logic transfer paths, the fault model for the data transfer function is independent of the actual implementation details of the transfer mechanisms.

Assumption 2. In the test generation algorithm, single functional fault within the group of sensitized RT-statements is assumed. But, we allow the presence of any number of faults as long as they don't mask the one currently under consideration.

The overall test generation algorithm is conceptually divided into three parts: pre-process, main-process (the S-algorithm), and post-process. The simplified interaction between these three parts is depicted in Figure 5.

5.1 Preprocess

The pre-process is mainly to perform syntax checking of the RT-description of the system-under-test, partition the whole RT-description into a set of ordered function submodules, and set up the basic data bases needed at later two stages. The order of test generation among the RT-statements in overall RT-description is a crucial issue. It is derived using the function submodules in the RT-description as the basic units to be ordered. A

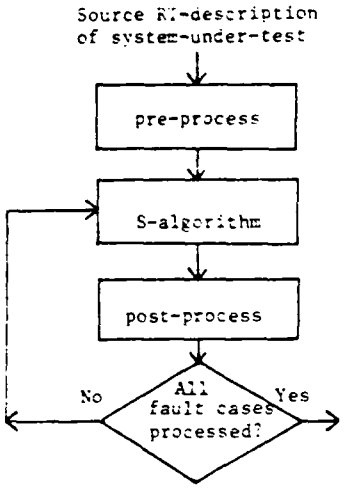


Figure 5 Simplified test generation flow

function submodule is a loop of path starting from the first RT-statement in the RT-description and has no branching path at the last node of the path right before the loop is formed. The logical meaning of function submodule in a general digital system is just like the "instruction" in a processor. The order of test generation of testable function submodules is set as follows:

first: pure-transfer vs. non-pure transfer
second: the number of distinct RT-statements
third: the number of distinct registers.

A function submodule with neither arithmatic operation nor logical operations is called a pure-transfer submodule. The start-small principle is applied in the derivation of the test order so that assumption 2 can be reasonably justified.

The steps performed in preprocess stage are shown below:

Step 1: Perform syntax checking of RT-description representing the system-under-test (SUT) and set up associated data bases.

Step 2: Derive all function submodules within the SUT.

Step 3: Prepare the order of test generation among all function submodules using the start-small principle.

Figure 6 shows the derived order of test generation among the eight function submodules in SIMPLE-CALCULATOR.

5.2 The S-algorithm

The "S" stands for "symbolic". In this stage, the RT-level symbolic execution technique is intensively employed for test generation of each fault modeled in the RT-level fault model. In the S-algorithm, the fault-free description of current function submodule (FS_i) is symbolically executed to set up a symbolic execution subtree (SET_i) which near-minimally covers all distinct RT-statements in FS_i. First, test inputs for data transfer faults (←/←'), except constant transfers (e.g., R_1←1), are computed using the transfer-test-finding heuristics and the symbolic results obtained at each terminated path. Faults in other fault types are enumerable and are

FS#	RT-statements in the loop	Is it pure-trans.	# of dist. RT-St.	# of dist. Reg.	Order of test gen.
1	0→1→2→3→11→10→0	N	2	4	6
2	0→1→2→3→4→12→10→0	N	2	4	7
3	0→1→2→3→4→5→13→14→15→16→17→18→19→20→0	N	9	9	8
4	0→1→2→3→4→5→6→21→22→10→0	Y	3	5	3
5	0→1→2→3→4→5→6→7→23→24→10→0	Y	3	5	4
6	0→1→2→3→4→5→6→7→8→25→26→10→0	Y	3	5	5
7	0→1→2→3→4→5→6→7→8→9→27→10→0	Y	2	3	2
8	0→1→2→3→4→5→6→7→8→9→10→0	Y	1	1	1

Figure 6 Function submodules in the SIMPLE-CALCULATOR

injected one at a time into the fault-free RT-description. For each fault case α, the symbolic execution subtree of the fault-injected machine (SET_i^{α}) is set up for a terminated path. The intermediate symbolic values along the fault-free path are saved and used to speed up the generation of such a path. An input test pattern for detecting this fault is then derived by comparing the symbolic results and path constraints of the fault-free and fault-injected machines. Four major issues must be considered in the S-algorithm: how to find the near-minimal covering of distinct RT-statements in the function submodules, the design of transfer-test-finding heuristics for data transfer faults, enumeration and identification of enumerable faults, and the solving of symbolic inequalities.

Using the assertions stated in Theorem 4 in Section 3, only all distinct RT-statements in each function sub-module FS_i need be considered. Therefore, not every symbolic execution path in FS_i need be traversed if a subset of symbolic execution paths which near-minimally covers all distinct RT-statements in FS_i can be found. The term "near-minimal" is used in the sense that the RT-statement covering is formed using heuristics rather than strict complex algorithm. The data transfer faults (←/←') is regarded as non-enumerable faults. The heuristics for transfer-test-finding are specially designed for the efficient generation of tests for such kinds of faults.

Except transfer faults, other faults need to be enumerated. After an enumerated fault is selected for test generation, it is injected into the corresponding RT-statement. The enumeration process exhaustively processes each enumerable fault under consideration. Solving symbolic inequalities is performed after the symbolic results and path constraints of the fault-free and the fault-injected machines under a fault α are obtained.

The steps executed by the S-algorithm are listed below:

Step 4: For a chosen function submodule FS_i of SUT, perform machine symbolic execution of FS_i's RT-description to set up a symbolic execution subtree SET_i which near-minimally covers all distinct RT-statements in FS_i by the path set $\{P_{ij}\}$.

Step 5: Perform heuristics of transfer-test-finding to find test patterns for data transfer faults in FS_i.

Step 6: Choose the next path P_{ij} in $\{P_{ij}\}$.

Step 7: Based on the RT-level fault model used, enumerate and inject a fault α which has not been tested along P_{ij}.

Step 8: Set up symbolic execution subtree for the fault-injected machine. Choose one terminated path P_{ij}^{α} for faulty symbolic results and path constraints.

Step 9: Derive test patterns for fault α by comparing the symbolic results and path constraints of P_{ij} and P_{ij}^{α}.

To show the process of test patterns generation, let us consider several typical illustrative examples of fault cases.

During the process of test generation of the simple-calculator, the RT-statement 11:F←A+B,→10 in "addition" submodule (submodule #1) will be tested along the path: 0→1→2→3→11→10→0.

Based on the conclusion of the RT-level fault collapsing analysis (Theorem 4.4 and 4.5), only the following fault types need be considered for this statement:

(1) +/+'
(2) R/R' with R∈{F, A, B}
(3) →12/→12'
(4) +/+'

where,(2), (3), (4) are enumerable faults.

Now, we consider one fault case out of each fault type for illustration.

(1) +/+' - In performing the Transfer-Test-Finding operation of this function submodule, the transfer paths of statement 11 must be considered are:

- transfer paths of registers A and B to ALU input ports.
- ALU output port to registers F and A.

Signal stuck-at-faults and bridging faults are considered in +/+' fault type.

① - Since before statement 11 is executed, the symbolic value of A is $A and B is $B, to test path of A to ALU input port, we must apply:

$A	$B
11111111	00000000
11110000	00000000
11001100	00000000
10101010	00000000
00000000	00000000

To test path of B to ALU input port, we must apply:

$A	$B
00000000	11111111
00000000	11110000
00000000	11001100
00000000	10101010
00000000	00000000

② - Since after ALU performed the addition, the result of ALU will be $A+$B, to test path of ALU output port to register F and A, we must apply:

A=10000000 B=10000000

in addition to those input test patterns in ①.

(2) R/R' with R∈{F, A, B}- The fault cases under this fault type are:

F'∈{AS, BS, OS, S} A'∈{B, 0} B'∈{A, 0}

Note that the fault cases of A' and B' are already considered in function submodules #4 and #5 and need not be tested again here.

Suppose we consider F'=AS, then the symbolic results of the fault-free machine are:
S=0 F=carry($A,$B) AS=$AS A=$A+$B, where $ is a symbolic value marker and the symbolic results of the fault-injected machine are:
S_α=0 F_α=$F AS_α=carry ($A,$B) A_α= $A+$B, where α is F/F' = F/AS both under the path constraints: S=1 E=001.

We then solve the set of algebraic inequality equations:

$$\begin{cases} S \neq S_\alpha \rightarrow & \text{¢} \\ F \neq F_\alpha \rightarrow & \text{carry ($A, $B)} \neq \$F = 0 \\ AS \neq AS_\alpha \rightarrow & 0=\$AS \neq \text{carry ($A, $B)} \\ A \neq A_\alpha \rightarrow & \text{¢} \end{cases}$$

Therefore, S=1 E=001 A=10000000 B=10000000 is a feasible test pattern for this fault case.

(3) →12/→12' - The fault cases under this fault type are:

12'∈{13=001101 8=001000 28=011100
14=001110 4=000100 44=101100}

Suppose we consider →12' =→13, then the symbolic results of the fault-free machine are: S=0 F=carry ($A,$B) A=$A+$B and the symbolic results of the fault-injected machine are: S_α=0 F_α=borrow ($A+$B-$B)=borrow($A)=$F=0 A_α=$A+$B-$B=$A, both under the path constraints: S=1 E=001. We then solve the set of algebraic inequality equations:

$$\begin{cases} S \neq S_\alpha \rightarrow & \text{¢} \\ F \neq F_\alpha \rightarrow & 0 \neq \text{carry ($A, $B)} \\ A \neq A_\alpha \rightarrow & \$A+\$B \neq \$A \text{ which turns out to be } \$B \neq 0. \end{cases}$$

Therefore, S=1 E=001 A=10000000 B=10000000 is a feasible test pattern for this fault case.

(4) +/+' - The fault cases under this fault type are: +'∈{-, XOR, shr}

Suppose we consider +' to be -, then the symbolic results of the fault-free machine are: S=0 F=carry ($A,$B) A=$A+$B and the symbolic results of the fault-injected machine are: S_α=0 F_α=borrow($A,$B) A_α=$A-$B, where α is +/+' = +/- both under the constraints: S=1 E=001.

We then solve the set of algebraic inequality equations:

$$\begin{cases} S \neq S_\alpha \rightarrow & \text{¢} \\ F \neq F_\alpha \rightarrow & \text{carry($A,$B)} \neq \text{borrow ($A,$B)} \\ A \neq A_\alpha \rightarrow & \$A+\$B \neq \$A-\$B \text{ which turns out to be } \$B \neq 0. \end{cases}$$

Therefore, S=1 E=001 A=10000000 B=10000000 is a feasible test pattern for this fault case.

5.3 Postprocess

The postprocess stage performs the following tasks:

1) Perform fault screening (elimination of covered faults from the fault list) using the test pattern just derived for a specific fault in stage two.
2) Repeat the S-algorithm if any unprocessed fault case remains.
3) Perform clean-up for hard-to-test faults.
4) Prepare the test generation report.

The process of fault screening is actually a kind of simulation which simulates the test pattern just derived in the S-algorithm on current fault-free path. Similar fault identification and numbering technique used in the S-algorithm will be needed here again.

When the steps in the S-algorithm are terminated, a set of global test patterns which has broad fault coverage for the entire system-under-test has been generated. Typically, a reasonable number of faults

would still be undetected. The difficult task now is to generate tests to "clean-up" those undetected faults. Since each individual fault case to be attacked by the clean-up operation is really "hard-to-test" (possibly due to the very complex hardware behavior or limitation of the test generation policy currently adopted), the combination of automatic test generation and human aids are both included in our current approach. The steps performed in the postprocess stage are:

Step 10: If no test pattern is obtained in Step 9, then return to Step 8 and try another P^2_{ij}; otherwise substitute this test pattern as an input data into P_{ij} and simulate all untested faults remaining on P_{ij}. Perform possible fault cases elimination along P_{ij}.

Step 11: If there is more fault cases left in P_{ij}, then return to Step 7.

Step 12: If there is more P_{ij} left in $FS_i(\{P_{ij}^1\})$, then return to Step 6.

Step 13: If there is more FS_i left in the SUT, then go to Step 4 and repeat.

Step 14: Perform possible clean-up operation for "hard-to-test" fault cases left which are indicated in the housekeeping tables.

Step 15: Prepare test generation report including test patterns obtained, their input sequence and other useful statistics.

6. DISCUSSION AND CONCLUSION

The complexity of the overall test generation algorithm in the last section is dominated mainly by the S-algorithm. A preliminary theoretic analysis of the S-algorithm shows that the complexity of the S-algorithm is dependent on the total number of RT-statements and the complexity of the symbolic execution system.

The S-algorithm developed has several analogies to the conventional gate-level D-algorithm. It has the following features:

1) For each testable RT-level fault, guarantee to find an input test pattern for detecting that fault.
2) Systematically find an input test pattern as early as possible.
3) Identify untestable RT-level faults.

The experimental prototype of the overall test generation algorithm is being implemented on the IBM 370/168-compatible main frame computer at SUNY-Binghamton. The preliminary experimental results are encouraging. More theoretical studies of complexity analysis of the S-algorithm and more solid experiments on several typical samples are being performed. By using valuable experience earned in software testing for hardware testing, this technique shows a promising way for future testing problems of digital LSI/VLSI systems.

REFERENCES

[1] E.I. Muehldorf and A.D. Savkar, "LSI logic testing - an overview", IEEE Trans. on Computers, Vol. C-30, No. 1, Jan 1981, pp. 1-17.

[2] S.Y.H. Su, "Computer-Aided design and testing of digital systems and circuits", (Invited paper), Proc. of the Jordan International Electrical & Electronics Engineering Conf., Amman, Jordan, Apr. 25-28, 1983, pp. 75-82.

[3] S.Y.H. Su and T. Lin, "Functional testing techiques for digital LSI/VLSI systems", (Invited paper), Proc. of 21st Design Automation Conf., June 1984, pp. 517-528.

[4] Shen and Su, "A functional testing method for microprocessors", Proc. of 14 Fault-Tolerant Computing Conference, June 1984, pp. 212-218.

[5] D. Brahme and J.A. Abraham, "Functional testing of microprocessors", IEEE Trans. on Computers, Vol. C-33, No. 6, June 1984, pp. 475-485.

[6] T. Lin, "Functional test generation of digital LSI/VLSI systems using machine symbolic execution technique", draft of Ph.D. dissertation, Computer Science Dept., State University of New York - Binghamton, Aug. 1984.

[7] J.A. Darringer and J.C. Kin, "Application of symbolic execution to program testing", COMPUTER, April 1978, pp. 51-60.

[8] L.A. Clarke, "A system to generate test data and symbolically execute programs", IEEE Trans. on software engineering, Vol. SE-2, No. 3, Sept. 1976, pp. 215-222.

[9] J.C. King, "A new approach to program testing", Proc. of International Conf. on Reliable Software, April 1975, pp. 228-233.

[10] R.S. Boyer, B. Elspas, and K.N. Levitt, "SELECT-A formal system for testing and debugging programs by symbolic execution", Proc. of International Conf. on Reliable software, April 1975, pp. 234-244.

[11] J.A. Darringer, "The application of program verification techniques to hardware verification", Proc. of 16th Design Automation Conference, 1979, pp. 375-381.

[12] W.C. Carter, W.H. Joyner Jr., and D. Brand, "Symbolic simulation for correct machine design", Proc. of 16th Design Automation Conf., 1979, pp. 280-286.

[13] S.Y.H. Su and Y.I. Hsieh, "Testing functional faults in digital systems described by register transfer language", Journal of Digital Systems, Vol. 6, No. 2, 1982, pp. 161-183. Also, Digest of papers, 1981 International Test Conf., pp. 447-457.

[14] J.D. Oakley, "Symbolic execution of formal machine description", Ph.D. thesis, Computer Science Dept., Carnegie-Mellon Univ., April 1979.

[15] S. Thatte and J. Abraham, "Test generation for microprocessors", IEEE Trans. on Computers, Vol. C-29, No. 6, June 1980, pp. 429-441.

END

FILMED

1-86

DTIC